Using Software Component Models and Services in Embedded Real-Time Systems

Frank Lüders, Shoaib Ahmad, Faisal Khizer, and Gurjodh Singh-Dhillon
Mälardalen University, Dept. of Computer Science and Electronics
PO Box 883, SE-721 23 Västerås, Sweden
frank.luders@mdh.se, {sad05004, fkr05001, gdn05001}@student.mdh.se

Abstract

While the use of software component models has become popular in the development of desktop applications and distributed information systems, such models have not been widely used in the domain of embedded real-time systems. Presumably, this is due to the requirements such systems have to meet with respect to predictable timing and limited use of resources. There is a considerable amount of research on component models for embedded real-time systems that focuses on source code components, statically configured systems, and relatively narrow application domains. This paper explores the alternative approach of using a mainstream component model based on binary components. The effects of using the model on timing and resource usage have been measured by implementing example applications both with and without using the model. In addition, the use of a prototype tool for supporting software component services has been investigated in the same manner.

1. Introduction

The use of software component models has become popular in the development of desktop applications and distributed information systems, where popular component models include *JavaBeans* [1] and *ActiveX* [2] for desktop applications and *Enterprise JavaBeans* (EJB) [3] and *COM+* [4] for information systems. In addition to basic standards for naming, interfacing, binding, etc., these models also define standardized sets of run-time services oriented towards the application domains they target. This concept is generally termed software component services [5].

Software component models havpe not been widely used in the development of real-time and embedded systems. It is generally assumed that this is due to the special requirements such systems have to meet, in particular with respect to timing predictability and limited use of resources such as memory and CPU time. Much research has been directed towards defining new component models for real-time and embedded systems, typically focusing on relatively small and statically configured systems. Most of the published research proposes models based on source code components and targeting relatively narrow application domains. Examples of such models include the *Koala* component model for consumer electronics [6], *PECOS* for industrial field devices [7], and *SaveCCM* for vehicle control systems [8].

An alternative approach is to strive for a component model for embedded real-time systems based on binary components and targeting a broader domain of applications, similarly to the domain targeted by a typical real-time operating system. This paper explores the possibility of using a mainstream component model as the starting point for such a model. Specifically, the use of the *Component Object Model* (COM) [9] with the real-time operating system *Windows CE* [10] is investigated. We have empirically evaluated the effect of using COM by implementing applications both with and without using the model. In addition, we have evaluated the effects of using a prototype tool for supporting software component services in embedded real-time systems.

The rest of this paper is organized as follows. Section 2 provides background information on COM and the prototype tool. Section 3 presents an automatic control applications that we use as an example to evaluate the use of these technologies. In Section 4, we described the tests we have conducted and their results. These results are discussed in Section 5. Section 6 is an overview of some related work. Conclusions and ideas for future work are presented in Section 7.

2. Background

2.1. The Component Object Model (COM)

Microsoft's Component Object Model (COM) [9] is one of the most commonly used software component models for desktop and server side applications. Although the model is increasingly being replaced by the newer .NET technology [11] in these domains, we believe COM is a more suitable starting point for a model aimed at embedded real-time systems because of its relative simplicity. In particular, the use of automatic memory management (garbage collection) in .NET is a serious barrier against ensuring predictable timing.

A key principle of COM is that interfaces are specified separately from both the components that implement them and those that use them. COM defines a dialect of the Interface Definition Language (IDL) that is used to specify object-oriented interfaces. Interfaces are object-oriented in the sense that their operations are to be implemented by a class and passed a reference to a particular instance of that class when invoked. The code that uses a component does not refer directly to any objects, however. Instead, the operations of an interface supported by an object are invoked via what is known as an interface pointer. A concept known as interface navigation makes it possible for the user to obtain a pointer to every interface supported by the object.

COM also defines a run-time format for interface pointers. What an interface pointer really references is an interface node, which in turn, contains a pointer to a table of function pointers, called a VTABLE. Typically, the node also contains a pointer to an object's instance data, although this is implementation specific. This use of VTABLEs is identical to the way that many C++ compilers implement virtual methods. Thus, the time and space overhead associated with accessing an object through an interface pointer is presumably the same as that incurred with C++ virtual methods. Figure 1 illustrates the typical format of interface nodes.

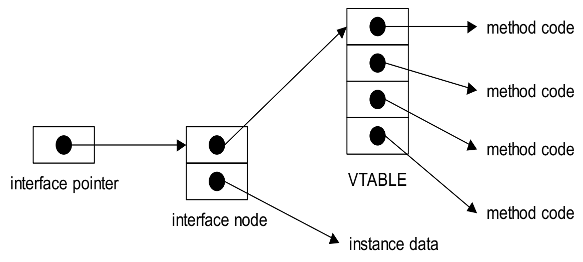


Figure 1. Typical format of COM interface nodes

For most real-time systems, a more serious concern than these modest overheads is that interface navigation introduces a possible source of run-time errors. If the user of a component asks an object for a pointer to an interface that the object does not support, this will not be detected during compilation. It may be argued, in fact, that this is the principal difference between interface navigation and interface inheritance in traditional object-oriented programming. This can be seen as a necessary price to pay for the otherwise desirable reduced compile-time dependence between components.

As already mentioned, a COM component is implemented in classes. The mechanism for creating instances of these classes is closely linked with how and when the code in different components is linked together. COM defines a policy for instantiation, which is intended to ensure that different components can be installed in a system at different times. When a component is installed, information about it must be registered somewhere in the system, linking the identity of its classes to the code that implement these. COM also requires a run-time library, called the COM library, to be installed on the system. When some code wants to use a component, it uses an operation provided by the COM library to ask for an instance of a class and an initial interface pointer to it. If the code of the component is not already loaded into memory, the COM library uses the registered information to locate the code and load it before an instance is created. This process is illustrated in Figure 2.

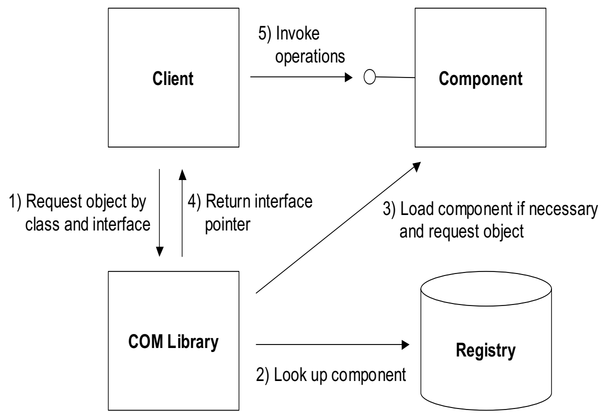

Client
5) Invoke operations
Component
1) Request object by class and interface
4) Return interface pointer
3) Load component if necessary and request object
COM Library
Registry
2) Look up component

Figure 2. Instance creation and dynamic loading of code in COM

Thus, creation of an instance involves searching the information about registered classes and possibly loading of code. This leads to a noticeable overhead when compared to instantiation in for instance C++. Furthermore, this overhead will vary, depending on whether the code implementing a class has already been loaded or not. This variability can be eliminated, however, by designing the software such that all components that may be used will be loaded at start-up. Note that removal of instances is subject to the same variability, since the COM standard states that code can be unloaded when the last instance that rely on it is removed.

A benefit that follows from COM's way of creating instances is that the code that implements a component can be built independently of any code that uses the component. Since instantiation involves passing the identity of the desired class as a parameter to a system operation, it is a possible source of run-time errors, which is not present during instantiation in traditional object-oriented programming, since attempting to instantiate a class that does not exist will result in a compilation error in this case. Again, this is a necessary price to be paid for decreased coupling.

2.2. Software component services for embedded real-time systems

A prototype tool for supporting software component services in embedded real time systems is presented in [12]. The tool adds services to COM components on Windows CE through the use of proxy object that intercept method calls. Figure 3 illustrates the use of a proxy object that provides a simple logging service. The object C2 implements an interface IC2 for which we wish to apply a logging service. A proxy object that also implements IC2 is placed between C2 and a client that uses the operations exposed through IC2. The operations implemented by the proxy forward all invocations to the corresponding operations in C2 in addition to writing information about parameter values, return codes, and invocation and return times to some logging medium.

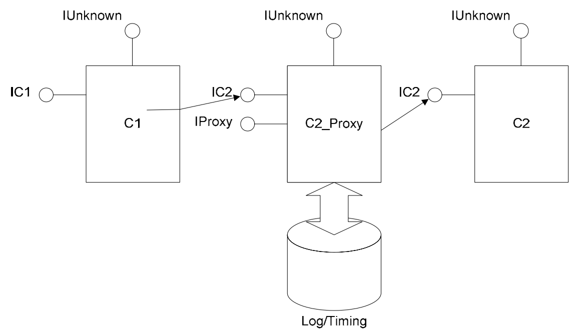


Figure 3. A logging service proxy

The tool takes as inputs a component specification along with specifications of desired services and generates source code for a proxy object. Component specifications may be in the form of Interface Definition Language (IDL) files or their binary equivalent Type Library (TLB) files. Desired services are either specified in a separate file using an XML-based format or in the tool´s graphical user interface, described further below. Note that access to component source code is not required. Based on these inputs, the tool generates a complete set of files that can be used with *Microsoft eMbedded Visual C++* to build a COM component implementing the proxy objects (i.e., the proxies are themselves COM objects). This process is depicted in Figure 4.

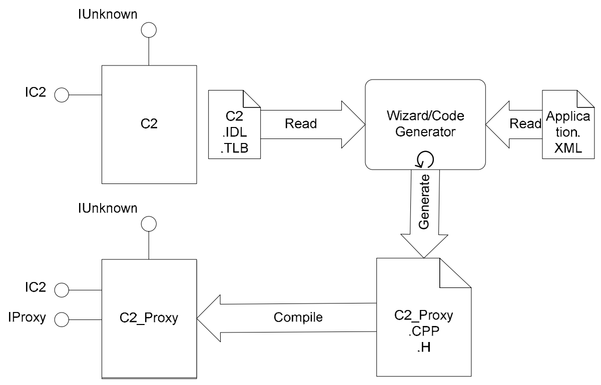
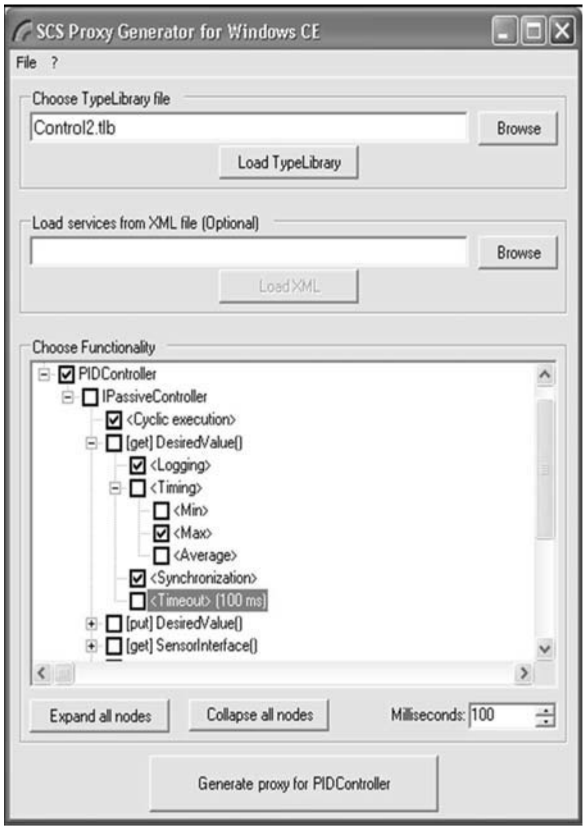


Figure 4. Proxy object generation

This use of proxy objects for interception is inspired by COM+. However, rather than to generate proxies at run-time, they are generated and compiled on a host computer and downloaded to the embedded system along with the application components. This process may occur when the software is initially downloaded to the system or as part of dynamic reconfiguration of a system that supports this. In the latter case, one can imagine updating or adding proxies without updating or adding any application components. The current version of the tool only generates proxy code and does not address the registration and run-time instantiation of components. This means that the client code must instantiate each proxy along with the affected COM object and set up the necessary connection between them.

In addition to logging, the tool supports generating proxies that implement one or more of the following services: execution time measurement of method invocations; synchronization between concurrent invocations; execution timeout on invocations; and cyclic execution of methods.

Figure 5 shows the graphical user interface of the tool. After a TLB or IDL file has been loaded all COM classes defined in the file are listed. Checking the box to the left of a COM class causes a proxy for that class to be generated when the button at the bottom of the tool is pressed. Under each COM class, the interfaces implemented by the class is listed and, under each interface, the operations implemented by the interface. In addition, the available services are listed with their names set in brackets. Checking the box to the left of a service causes code to be generated that provides the service for the element under which the service is listed. In the current version of the tool, a service for cyclic execution may only be specified for the IPassiveController interface while all other services may only be specified for individual operations. The IPassiveController interface is described in connection with the example application in the next section.



Control2.tlb

```
⊟ ☑ PIDController
   ⊟ ☐ IPassiveController
      ☑ <Cyclic execution>
      ⊟ ☐ [get] DesiredValue()
         ☑ <Logging>
         ⊟ ☐ <Timing>
            ☐ <Min>
            ☑ <Max>
            ☐ <Average>
         ☑ <Synchronization>
         ☐ <Timeout> (100 ms)
      ⊞ ☐ [put] DesiredValue()
      ⊞ ☐ [get] SensorInterface()
```

Milliseconds: 100

Figure 5. User interface of the prototype tool

If the cyclic execution service is checked, the proxy will implement an interface called IActiveController instead of IPassiveController (see the example in the next section). IActiveController includes operations for setting the period and threading priority of the cyclic execution. Checking the logging service results in a proxy that logs each invocation of the affected operation. The timing service causes the proxy to measure the execution time of the process and write it to the log at each invocation (if timing is checked but not logging, execution times will be measured but not saved).

The synchronization service means that each invocation of the operation will be synchronized with all other invocations of all other operations on the proxy object for which the synchronization service is checked. The only synchronization policy currently supported is mutual exclusion. The timeout service has a numeric parameter. When this service is selected (by clicking the name rather than the box) as in Figure 5, an input field marked Milliseconds is visible near the bottom of the tool. Checking the service results in a proxy where invocations of the operation always terminate within the specified number of milliseconds. In the case that the object behind the proxy does not complete the execution of the operation within this time, the proxy forcefully terminates the execution and returns en error code.

3. Example application

To evaluate the effects of using both COM and the prototype tool, we used the example application presented in [12]. At the center of this application is a component that encapsulates a *proportional-integral-differential* (PID) controller [13]. Four different versions of the application were implemented. They are presented here in the order in which they were first developed. The four versions are summarized in Table 1 at the end of this section.

We first implemented a version using COM, shown in Figure 6, which we term Control2. PIDController is a COM class that implements an interface IActiveController and relies on the two interfaces ISensor and IActuator to read and write data from/to the controlled process. For the purpose of this example, these interfaces are implemented by the simple COM class DummyProcess that does nothing except returning a constant value to the controller. The interfaces are defined as follows:

```
interface ISensor : IUnknown
{
  [propget] HRESULT ActualValue(
    [out, retval] double *pVal);
};

interface IActuator : IUnknown
{
  [propget] HRESULT DesiredValue(
    [out, retval] double *pVal);
  [propput] HRESULT DesiredValue(
    [in] double newVal);
};

interface IController : IActuator
{
  [propget] HRESULT SensorInterface(
    [out, retval] ISensor **pVal);
  [propput] HRESULT SensorInterface(
    [in] ISensor *newVal);
  [propget] HRESULT ActuatorInterface(
    [out, retval] IActuator **pVal);
  [propput] HRESULT ActuatorInterface(
    [in] IActuator *newVal);
  [propget] HRESULT CycleTime(
    [out, retval] double *pVal);
  [propput] HRESULT CycleTime(
    [in] double newVal);
  [propget] HRESULT Parameter(
    [in] short Index,
    [out, retval] double *pVal);
  [propput] HRESULT Parameter(
    [in] short Index,
    [in] double newVal);
};

interface IActiveController
  : IController
{
  [propget] HRESULT Priority(
    [out, retval] short *pVal);
  [propput] HRESULT Priority(
    [in] short newVal);
  HRESULT Start();
  HRESULT Stop();
};
```

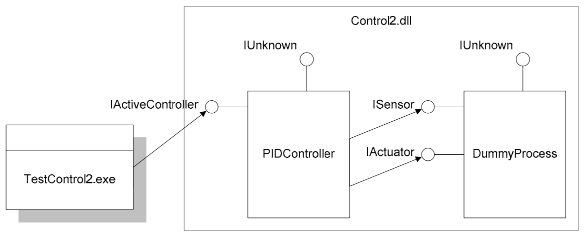


Figure 6. Implementation with COM

IController is a generic interface for a single-variable controller with configurable cycle time and an arbitrary number of control parameters. PIDController uses three parameters for the proportional, integral, and differential gain. IActiveController extends this interface to allow control of the controller´s execution in a separate thread. (The reason for splitting the interface definitions like this was to reuse IController for a controller that uses the cyclic execution service rather than maintaining its own thread.) Note that IController inherits the DesiredValue property from IActuator. This definition was chosen to allow the interface to be used for cascaded control loops where the output of one controller forms the input to another.

The test application TestControl2.exe creates one instance of PIDController and one instance of DummyController. It then connects the two objects by setting the SensorInterfaca and ActuatorInterface properties of the PIDController object. After this it sets the cycle time and the control parameters before invoking the Start operation. This causes the PIDController object to create a new thread that executes a control loop. A simple timing mechanism is used to control the execution of the loop in accordance with the cycle time property. At each iteration the loop reads a value from the sensor interface, which it uses in conjunction with the desired value, the control parameters, and an internal state based on previous inputs to compute and write a new value to the actuator interface. To minimize jitter (input-output delay as well as sampling variability), this part of the loop uses internal copies of all variables, eliminating the need for any synchronization.

Next, the control loop updates its internal variables for subsequent iterations. Since the desired value and the control parameters may be changed by the application while the controller is running, this part of the loop uses a mutual exclusion mechanism for synchronization. In addition to performing its control task the loop timestamps and writes the sensor and actuator data to a log. The control loop is illustrated by the following pseudo code:

```
while (Run) {
   WaitForTimer();
   ReadSensorInput();
   ComputeAndWriteActuatorOutput();
   WriteDataToLog();
   WaitForMutex();
   UpdateInternalState();
   ReleaseMutex();
}
```

Note that, due to the simple timing mechanism, the control loop will halt unless all iterations complete within the cycle time.

Next, we implemented a component intended to perform the same function, but relying on services provided by generated proxies. A test application using this component and proxies is shown in Figure 7. In this application, termed Control3, PIDController is a COM class that implements the IPassiveController interface. Note that, although this COM class has the same human readable name as in the application described above, it has a distinct identity to the COM run-time environment. To avoid confusion we use the notation Control3.PIDController when appropriate. IPassiveController extends IController as follows:

```
Interface IPassiveController
  : IController
{
  HRESULT UpdateOutput();
  HRESULT UpdateState();
};
```

These operations are used by the proxy_PIDController object to implement a control loop that performs the same control task as in the previous example.

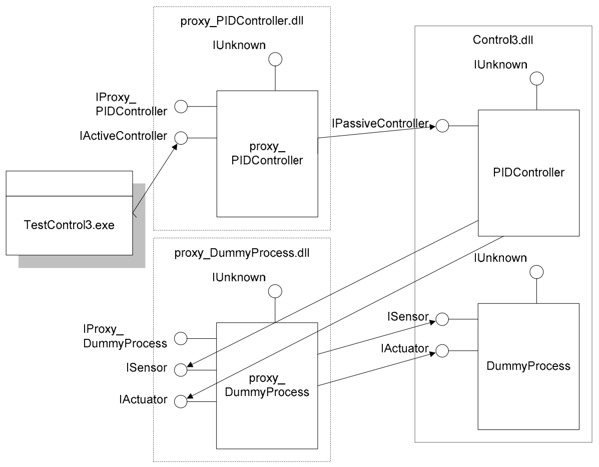


Figure 7. Implementation with COM and generated proxies

The proxy_PIDController COM class was generated with the use of the tool by checking the cyclic execution service under the Control3.PIDController´s IPassiveController interface. The proxy_DummyProcess COM class provides the interface pointers for the controller's SensorInterface and ActuatorInterface properties. Behind this proxy is a DummyProcess object with the same functionality as in the Control2 application. proxy_DummyProcess was generated by the tool with the logging service checked. As a result, all data read and written via the sensor and actuator interfaces are logged. The interfaces IDummyProcess_Proxy and IPIDController_Proxy are only used to set up the connections between proxies and other objects. They are defined as follows:

```
interface IProxy_DummyProcess
  : IUnknown
{
  HRESULT AttachISensor(
    [in] IUnknown *pTarget);
  HRESULT AttachIActuator(
    [in] IUnknown *pTarget);
};

interface IProxy_PIDController
  : IUnknown
{
  HRESULT AttachIPassiveController(
    [in] IUnknown *pTarget);
};
```

To be able to evaluate the overhead introduced by the use of COM and the generated proxies, we implemented two non-component-based versions of the application, each consisting of a single executable file. Figure 8 shows the internal structure of these programs, termed Control0 and Control1, as UML class diagrams.

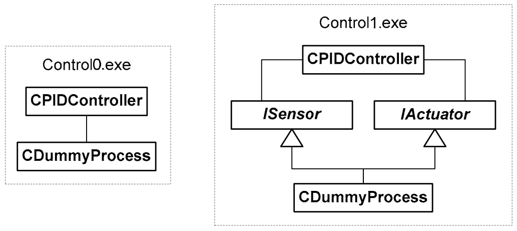


Figure 8. Non-component-based implementations

The application termed Control1 was constructed by making very modest modifications to the source code of the Control2 application. The main modification was that the calls to the COM library for creating instances of COM classes were replaced by simple instantiation of C++ classes. The C++ classes CPIDController and CDummyProcess are identical to those used internally to implement the COM classes of Control2. ISensor and IActuator are abstract C++ classes that correspond directly to the COM interfaces of the same names. They are specified in C++ as follows:

```
class ISensor : public IUnknown
{
  virtual HRESULT get_ActualValue(
    double *pVal) = 0;
};

class IActuator : public IUnknown
{
  virtual HRESULT get_DesiredValue(
    double *pVal) = 0;
  virtual HRESULT put_DesiredValue(
    double newVal) = 0;
};
```

Control0 is a modified version of Control1, where the classes are modified such that virtual methods are not used. This means that calls to the methods are not performed using VTABLES of function pointers, and the address of the methods are determined at compile-time rather than at run-time. The abstract classes are removed, since such classes rely entirely on virtual methods. Table 1 summarizes the four different versions of the application.

Table 1. Summary of application versions

Name	Description
Control0	Using C++ without virtual methods
Control1	Using C++ with virtual methods
Control2	Using COM
Control3	Using COM and proxy-based services

4. Tests

4.1. Test setup

The example application described in the previous section was tested on a system running Window CE 5.00. The hardware used was a PC with a 2.8 GHz Pentium 4 processor. The Windows CE run-time image was built using Microsoft Platform Builder 5.00 with the standard board support package for a Windows CE based PC (CEPC) and the standard setting provided by the "Industrial Controller" platform template. This platform allowed time measurements to be made with a resolution of one millisecond. Each of the four versions of the application was built with Microsoft eMbedded Visual C++ and tested on the target computer one at a time, resetting the target between each test.

For each of the four versions of the example application, two different execution times were measured. The first was the time required for invocation of the get_ActualValue method of the DummyProcess COM objects or, in the case of Control0 and Control1, of the CDummyProcess C++ objects. Given the one millisecond resolution, we were required to modify the control loop of the programs by adding an inner loop that performed two million invocations of get_ActualValue instead of a single invocation to obtain usable time measurements. For each of the versions, this measurement was made 170 times.

The second measurement made for each of the versions was the time required for initialization of the application. This initialization includes instantiation of the COM or C++ objects and setting up of the connections between them. This test was performed 20 times for each of the versions of the example application.

In addition to execution times, measurements of memory usage were also performed. However, we were not able to see any difference between the four different versions of the test application on the test platform we used. Also, differences between the size of source code and binary files were presented in [12] and are not repeated here. Thus, the following presentation and discussion of the results focus on execution times.

4.1. Results

Figure 9 shows measured execution times of making two million invocations of get_ActualValue for the four different version of the example application. The measurements for Control0 (without COM and not using virtual methods) are the lowest with an average of four milliseconds. These measurements show no variation, but given that the resolution is one millisecond the uncertainty per measurement is 25%.

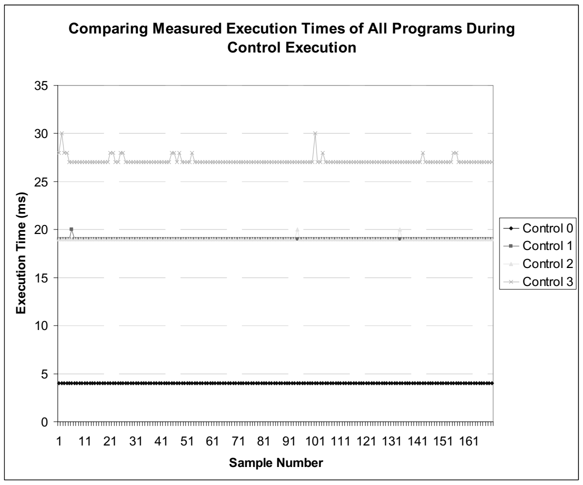


Figure 9. Measured execution times

Control1 (without COM but using virtual methods) and Contro2 (with COM) give similar results of approximately 19 milliseconds on average and 5% variation. This indicates that the overhead of using COM as well as of using virtual methods in C++ is approximately 15 milliseconds. Taking into account that two million invocations were made per measurement, this correspond to an invocation overhead of 7.5 nanoseconds for this particular processor.

Control3 (with COM and all invocations passing through a proxy objects) gives approximately 27 milliseconds on average and 11% variation. This indicates an additional overhead of approximately eight milliseconds compared to Control2, corresponding to four nanoseconds per invocation. Table 2 summarizes the measurements depicted in Figure 9.

Table 2. Summary of execution times

Version	Execution time (ms)		
	Min.	Max.	Average
Control0	4	4	4
Control1	19	20	19.00588
Control2	19	20	19.01176
Control3	27	30	27.12353

Figure 10 shows measured execution times of application initialization for Control0, Control1, and Control2. The measurements for Control0 (where the initialization consists of instantiating two C++ classes and passing a reference of one instance to the other) give an average of 0.4 milliseconds. For Control1 (where the initialization is very similar) the average is 0.7 milliseconds and for Control2 (where initialization involves calling the COM library to instantiate the COM classes) one millisecond. Given that these values are so small compared to the one millisecond resolution and that only 20 measurements were collected in each case, they can only be viewed as crude estimations of the real execution time.

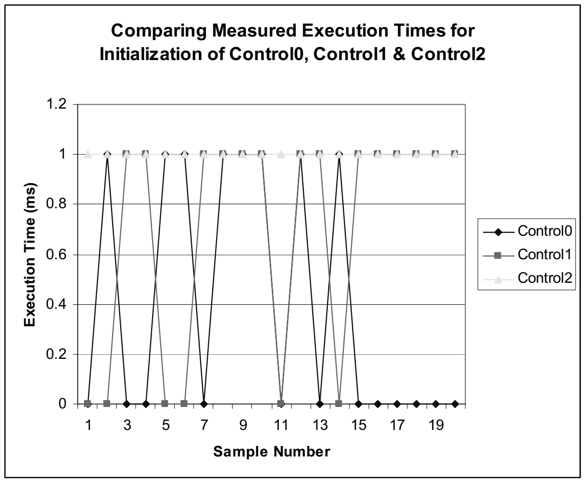


Figure 10. Measured initialization times for Control0, Control1, and Control2

Figure 11 shows measured execution times of application initialization for Control3. For this implementation (where the initialization comprises calling the COM library to instantiate four different COM classes in three different components and performing a comparatively complex setup task) the average is approximately 2940 milliseconds, which is off course notably higher than for the other implementations. The variation is also quite high with a difference of 4686 milliseconds between the minimum and maximum. If we treat the maximum value as an outlier we get approximately 2730 milliseconds on average and 40% variation. The measured execution times of application initialization is summarized in Table 3.

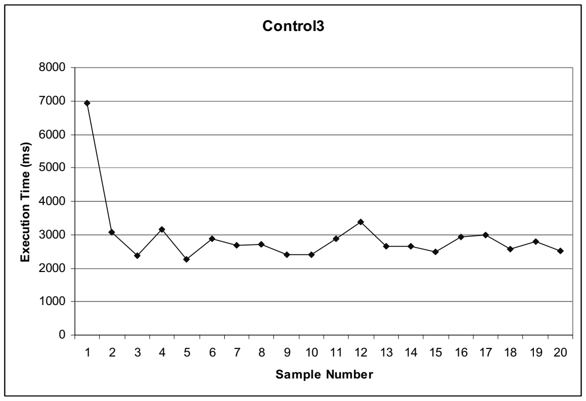


Figure 11. Measured initialization times for Control3

Table 3. Summary of initialization times

Version	Execution time (ms)		
	Min.	Max.	Average
Control0	0	1	0.4
Control1	0	1	0.7
Control2	1	1	1
Control3	2264	6950	2940.85

5. Discussion

The measured execution times during control execution constitute quite strong evidence that the overheads of using both COM and the proxy-based services are modest and, even more importantly in many real-time systems, quite predictable. The overhead of using COM interfaces pointers are found to be essentially equal to that of using C++ virtual methods. It should be noted here, however, that using C++ classes allows mixing of virtual and statically bound methods.

Typically, a carefully designed C++ class will use virtual methods only where variability is desired, leading to a lower average overhead than for an equivalent COM class where invocation through interface pointers is mandatory.

Comparing the use of proxy-based services with the use of COM without services, the additional overhead is found to be quite modest. In some cases, however, the increased number of indirections might be expected to lead to an increase in the number of cache misses and thereby a higher penalty. Clearly, the tests presented in this paper, where a single operation has been invoked repeatedly in a loop, is much less likely to result in cache misses than more realistic usage scenarios.

Based on the measured execution times during application initialization, it seams safe to conclude that the times required when C++ or COM is used are of the same order of magnitude, while the time required when proxy-based services are used are several orders of magnitude higher. Nonetheless, these measurements leave much to be desired. For Control0, Control1, and Control2, it would be desirable to perform additional measurements using loops that repeat the initializations to obtain higher values and hence increased accuracy. For Control3, additional measurements to reveal the most time consuming parts of the initialization phase would be very desirable.

6. Related work

Although models based on source code component still seem to dominate, there are other efforts to support binary software components for embedded real-time systems. One example is the ROBOCOP research project [14], which builds on the aforementioned Koala component model and primarily targets the consumer electronics domain. The component model defined as part of this project is largely based on the basic concepts of COM. Furthermore, the sequel of the project, called Space4U [15], also seems to use a mechanism similar to software component services, e.g. to support fault-tolerance.

The approach to software component services discussed in this paper relies heavily on the technique of providing services by interception. This mechanism is also used in other technologies and is sometimes called interceptors rather than proxies, e.g. in the *Common Object Request Broker Architecture* (CORBA) [16] and the MEAD framework for fault-tolerant real-time CORBA applications [17]. The approach is furthermore similar to the concept of aspects and weaving. In [18], a real-time component model called *RTCOM* is presented which have support for weaving of functionality into components as aspects while maintaining real-time properties. An important difference with our approach is that, in RTCOM, functionality is weaved in at the level of source code..

Another effort towards adapting a mainstream component model to the embedded real-time systems domain is presented in [19]. This work aims to extend the Enterprise JavaBeans model with means for specifying timing properties of software components. As it focuses on specification and not run-time issues, it is complementary to our work rather than an alternative. The fact that it is based on EJB rather than COM is not of principal importance, but the lack of Java run-time environments for embedded real-time systems may mean that the approach is further from real-world application.

In general, the concept of software component services can be seen as a special case of middleware. The use of middleware in embedded real-time systems is an active topic of research (and practice) not necessarily related to software components. Similar to our approach of adapting a mainstream component model, efforts have been made to adapt mainstream middleware to the domain of embedded real-time systems [20]. Specialized middleware frameworks for this domain also exist, including *OSA+* [21] that provides services for distributed systems and *Kokyu* [22] that provides flexible scheduling and dispatching services.

7. Conclusion and future work

The aim of the work presented in this paper has been to investigate the possibility of using a mainstream software component model, as well as an extension of this model with run-time services, for developing embedded real-time systems. We believe that the results show that this is a promising approach, although further investigation, in particular of the overheads related to object instantiation, should be undertaken. The overheads related to invoking operations through COM interfaces as well as through a forwarding proxy were found to be both modest and predictable. Thus, these overheads would probably be quite acceptable in many embedded real-time systems. In particular, we can conclude that a system that can afford the invocation overhead of C++ virtual methods can also afford COM interfaces, since the cost is nearly identical.

Since we view the use of a mainstream component model like COM as an alternative to more specialized models, it would be interesting to conduct a comparative study of COM (and possibly other mainstream models) and at least one specialized model. A possible object of such a study is the aforementioned Koala component model, which is supported by freely available tools. Differences between models, e.g. in terms of time and memory overheads, should be investigated empirically by implementing example applications.

In addition to the run-time effects on resource usage and predictability, the effects of using the approach on development effort and such quality attributes as reliability and reusability should be evaluated. In our future work, we aim to do this using different empirical techniques, including both controlled experiments and case studies with student participation. In addition, it would be desirable to perform industrial case studies, which implies a lower level of control and replication, but allows more realistic situations to be investigated.

8. References

[1] R. Englander, *Developing Java Beans*, O'Reilly, 1997.

[2] D. Chappell, *Understanding ActiveX and OLE*, Microsoft Press, 1996.

[3] R. Monson-Haefel, B. Burke, and S. Labourey, *Enterprise JavaBeans*, 4th edition, O'Reilly, 2004.

[4] D.S. Platt, *Understanding COM+*, Microsoft Press, 1999.

[5] G.T. Heineman and W.T. Council, *Component-Based Software Engineering – Putting the Pieces Together*, Addison-Wesley, 2001.

[6] R. van Ommering, F. van der Linden, and J. Kramer, "The Koala Component Model for Consumer Electronics Software", *Computer*, volume 33, issue 3, 2000.

[7] T. Genßler, C. Stich, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, and P. Müller, "Components for Embedded Software – The PECOS Approach", *Proceedings of the 2002 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, 2002.

[8] H. Hansson, M. Åkerholm, I. Crnkovic, and M. Törngren, "SaveCCM – A Component Model for Safety-Critical Real-Time Systems", *Proceedings of the 30th EROMICRO Conference*, 2004.

[9] D. Box, *Essential COM*, Addison-Wesley, 1997.

[10] J. Murray, *Inside Microsoft Windows CE*, Microsoft Press, 1998.

[11] D.S. Platt, *Introducing Microsoft .NET*, 3rd edition, Microsoft Press, 2003.

[12] F. Lüders, D. Flemström, A. Wall, and I. Crnkovic, "A Prototype Tool for Software Component Services in Embedded Real-Time Systems", *Proceedings of the 9th International Symposium on Component-Based Software Engineering*, 2006.

[13] K.J. Åström and B. Wittenmark, *Computer Controlled Systems – Theory and Design*, 2nd edition, Prentice Hall, 1990.

[14] J. Muskens, M.R.V. Chaudron, and J.J. Lukkien, "A Component Framework for Consumer Electronics Middleware", C. Atkinson, C. Bunse, H. Gross, and C. Peper (editors.), *Component-Based Software Development for Embedded Systems*, Springer, 2005.

[15] Space4U Project, "Space4U Public Homepage", 2006, http://www.hitech-projects.com/euprojects/space4u/, Accessed on 28 April 2006.

[16] Object Management Group, "Common Object Request Broker Architecture – Core Specification", OMG formal/04-03-12, March 2004.

[17] P. Narasimhan, T.A. Dumitras, A.M. Paulos, S.M. Pertet, C.F. Reverte, J.G. Slember, and D. Srivastava, "MEAD – Support for Real-Time Fault-Tolerant CORBA", *Concurrency and Computation – Practice and Experience*, volume 17, issue 12, February 2005.

[18] A. Tesanovic, D. Nyström, J. Hansson, and C. Norström, "Aspects and Components in Real-Time System Development – Towards Reconfigurable and Reusable Software", *Journal of Embedded Computing*, volume 1, number 1, February 2004.

[19] S. Wang, S. Rho, Z. Mai, R. Bettati, and W. Zhao, "Real-Time Component-Based Systems", *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium*, 2005.

[20] D.C. Schmidt, "Middleware for Real-Time and Embedded Systems", *Communications of the ACM*, volume 45, issue 6, June 2002.

[21] F. Picioroaga, A. Bechina, U. Brinkschulte, and E. Schneider, "OSA+ Real-Time Middleware, Results and Perspectives", *Proceeding of the 7th International IEEE Symposium on Object-Oriented Real-Time Distributed Computing*, 2004.

[22] C.D. Gill, R.K. Cytron, and D.C. Schmidt, "Multiparadigm Scheduling for Distributed Real-Time Embedded Computing", *Proceedings of the IEEE*, volume 91, issue 1, January 2003.